NetSpec
User Manual

Anupama Sundaresan
anu@ittc.ukans.edu

April 1999

Author of NetSpec
Roel Jonkman

Authors of NetSpec Daemons
Beng-Ong Lee, Yulia Wijata, Sandeep Bhat, Anil Gopinath, Shyam Murthy

Information and Telecommunication Technology Center
University of Kansas
Lawrence, KS 66045

Web Site: http://www.ittc.ukans.edu/netspec

Abstract

Development and deployment of new networking technologies initiated the development of appropriate testing tools. Network measurement has so far been dominated by passive monitoring. Also the existing testing and performance analysis tools have proven to be very limited in their abilities. Due to a need for active testing tools, NetSpec was developed. To cater to a wide variety of testing and measurement requirements, a number of daemons have been developed under the NetSpec framework. This user manual is an attempt to highlight the features and limitations of NetSpec. This document describes the usage of the different daemons and is an attempt to shed light on how to use NetSpec to conduct network measurement experiments.

Table of Contents

1.0 Introduction to NetSpec 5
1.1 Features of NetSpec 6
1.2 NetSpec Architecture 7
2.0 NetSpec Daemon Structure and Protocols 8
2.1 Execution Constructs 9
3.0 Running NetSpec 11
3.1 Purpose of Scripts 13
4.0 NetSpec Daemons 14
4.1 Test Daemon (nstestd) 14
4.1.1 Description of user NetSpec Scripts and Examples 14
4.1.2 Full Stream Traffic Scripts 15
4.1.3 Burst Traffic scripts 25
4.1.4 Queued Burst Traffic scripts 31
4.1.5 Emulated Traffic types and scripts 37
4.2 Data Stream Daemon (nsdstrd) 44
4.2.1 Features 44
4.2.2 NetSpec Script 44
4.2.3 Examples 45
4.3 SNMP Daemon (nssnmpd) 48
4.3.1 NetSpec Script 48
4.3.2 Example 49
4.4 CORBA Daemon (nscorbad) 50
4.4.1 Example 51
4.5 Private Network – Network Interface Daemon (pnnid) 54
4.5.1 Features of PNNI 54
4.5.2 NetSpec Script 55
4.6 Switch Daemon (nsswitchd) 58
4.6.1 NetSpec Script 58
4.6.2 Description of Parameters 58
4.7 Host Daemon (nsabrd) 61
4.7.1 NetSpec Script 61
5.0 Appendix 64
5.1 Installation of NetSpec 64
5.1.1 inetd Installation 64
5.1.2 Standalone Installation (i.e. you don't have root access) 64
5.2 Working of NetSpec 65
5.3 Tables 66
5.3.1 Table 10 Explanation of NetSpec Script keywords 66
5.3.2 Table 11 Keywords used in Emulated Traffic Scripts 68
6.0 Bibliography 69

List of Figures

Figure 1 NetSpec 3.0 Architecture 7
Figure 2 The Three types of Execution Constructs 10
Figure 3 Test setup with the invoked daemons over TCP 16
Figure 4 Test Setup over UDP 20
Figure 5 Architecture of a PNNI Emulation 55

List of Tables

Table 1 Different Phases of execution of the Test performed 9
Table 2 Description of the parameters 49
Table 3 List of parameters in the Script and their Description 50
Table 4 List of Test Types 51
Table 5 Allowable Parameters of parameter type " switch " 58
Table 6 Allowable Parameters of parameter type " connection " 59
Table 7 Allowable Parameters for "atmFull" and "atmSink" 62
Table 8 Parameter Set for PVC 62
Table 9 Parameter Set for SVC 63
5.3.1 Table 10 Explanation of NetSpec Script keywords 66
5.3.2 Table 11 Keywords used in Emulated Traffic Scripts 68

1.0 Introduction to NetSpec

The tremendous growth in the field of networking technologies initiated the development of appropriate testing tools. Development and deployment of advanced techniques in this field have necessitated the need for the right kind of testing tools. Network management has so far been dominated by passive monitoring. Emerging networking technologies however force the development of active testing and performance analysis tools. As a result of the limitations of the existing testing and performance analysis tools, NetSpec was developed.

NetSpec is a network level end-to-end performance evaluation tool developed by researchers in the University of Kansas, for the ACTS ATM Internetwork (AAI) project. The NetSpec system provides support for large-scale data communication network performance tests with a variety of traffic source types and modes. This software tool provides a simple block structured language for specifying experimental parameters and support for controlling performance experiments containing an arbitrary number of connections across a LAN or WANS.

NetSpec exhibits many features like parallel and serial multiple connections, a range of emulated traffic types (FTP, HTTP, MPEG, etc.) on the higher levels, the most widely used transport protocols today, that is TCP and UDP, three different traffic modes, scalability, and the ability to collect system level information from the communicating systems as well as intermediate network nodes. The most frequently used performance tools (ttcp, Netperf) do not support most of these features.

NetSpec can run on a variety of platforms. It has been successfully ported to the most commonly used platforms like DEC Alpha DUnix 3.0 3.2 and 4.0, SunOs, Solaris, Linux2.0.x and IRIX 5.3 and 6.1. Also NetSpec supports a wide variety of test/measurement daemons in its framework. This enables the user to select the daemon based on his requirements. Thus NetSpec provides the user with a capability to run tests on a variety of architectures and also helps the users to perform different kinds of tests to gather different types of network measurement information.

1.1 Features of NetSpec

- **Scalability**

The framework that is provided is scalable to carry out multiple tests. This is very desirable since networks generally carry hundreds of flows simultaneously.

- **Flexibility**

The framework is flexible incorporating both passive(probes, measurements) and active nodes (traffic generators).

- **Reproducible**

The results of the network are reproducible, assuming that the state of the network did not change.

- **Integration**

Measurements and tests are integrated in a seamless manner.

- **Extensibility**

The design has been done with a provision to add new components. New daemons can be incorporated in the existing framework with ease.

1.2 NetSpec Architecture

Figure 1 shows the basic NetSpec 3.0 architecture. The controller is a process that supports the user interface. The connection is the basic unit for an experiment and the control daemon controls the daemons implementing the test. For every connection in the experiment, the corresponding measurement/test daemons are created. These measurement/test daemons concentrate on performing the traffic related tasks (send or receive data transferred across the connection). Each daemon is responsible for its own report generation after experiment execution is complete and measurement daemons concentrate on collecting data as accurately as possible, without having to worry about performing traffic functions. The output report is delivered to the controller via the control daemon for viewing by the user. The communication between the controller and the daemons is achieved using an ASCII based language, which enhances portability and extensibility.

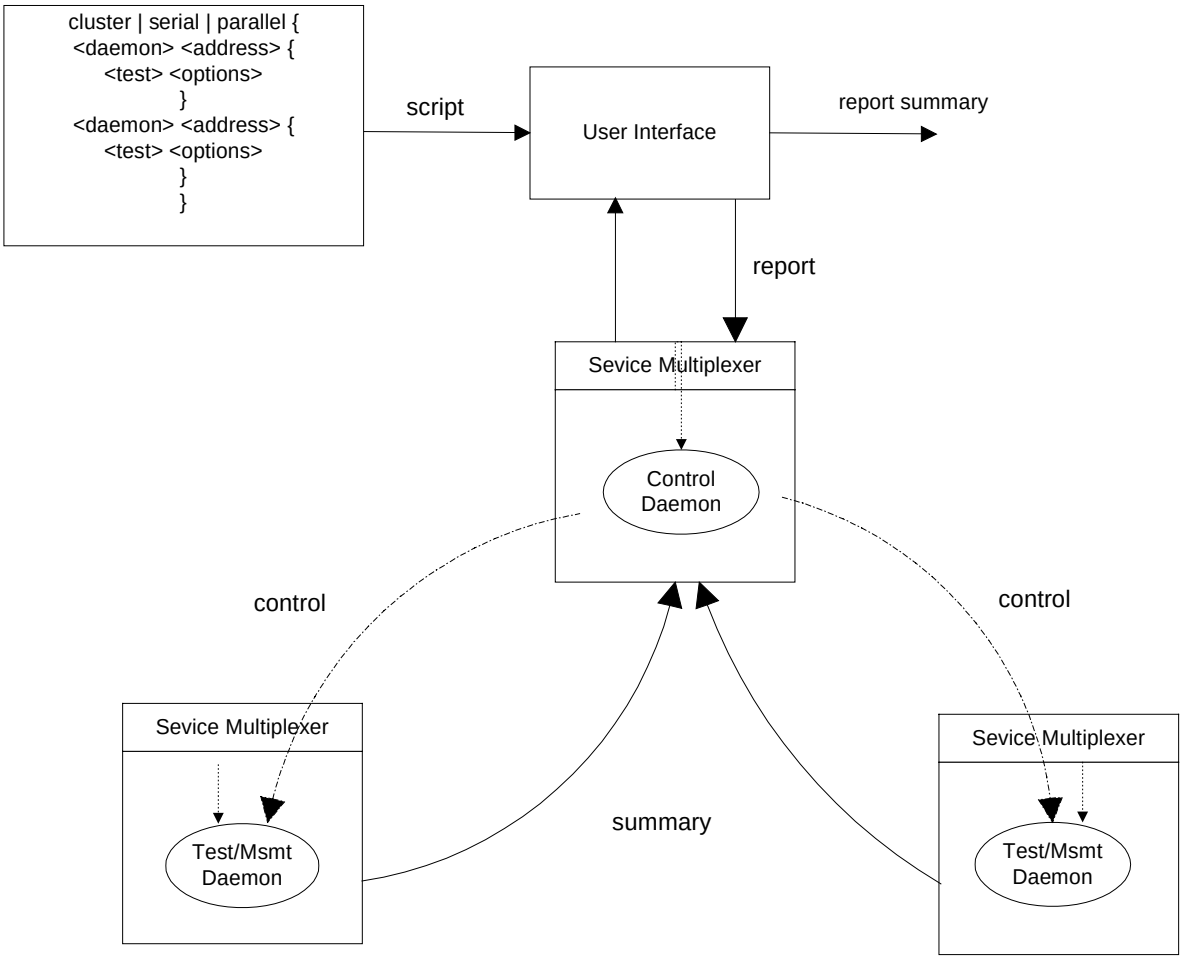

Figure 1 NetSpec 3.0 Architecture

- User Interface

  - Reads the script and passes it on to the Control Daemon
  - Generates the protocol commands for the daemon invocation
  - Accepts the summaries of the experiments and present to the user

- Service Multiplexer

  - Provides a single access point for invocation of all NetSpec daemons

- Control Daemon

  - Parses control protocol and parameters
  - Controls the invocation of other NetSpec daemons

- Measurement/Test Daemon

  - performs specific measurement or testing functionality

2.0 NetSpec Daemon Structure and Protocols

Since the primary objective of the control framework is to control traffic sources and sinks the control protocol is tailored to execute network connections with as precise control as possible. NetSpec provides a generic structure for a daemon implementation. Both the controller and the measurement/test daemon are equivalent from a connection and protocol perspective.

NetSpec uses a text-based protocol to control the execution of the nodes involved in an experiment. The Remote Control and Information Protocol (RCIP) imposes some sequence of phases undergone by each daemon during execution. These phases mimic the phases of a typical network connection. In addition, RCIP also supports some administrative commands for control purposes. The control daemon has a custom RCIP handler, which is responsible for distributing control across multiple nodes. The table below summarizes the commands supported by RCIP and a brief description of the actions taken when a command is invoked.

Table 1 Different Phases of execution of the Test performed

Command	Action
Setup	Allocate resources
Open	Establish a connection
Run	Start data transfer or measurement
Finish	Finish data transfer or measurement
Close	Close connection
Tear down	Free up resources
Report	Send report summary
Reset	Reset to initial state
Kill	Stop execution of daemon
Parameters	Accept parameters
Config	Return configuration information

The five phases of the connection as such dominate the protocol that is used to control the sources and sinks. There is a difference between unreliable and reliable protocols. In reliable protocols like TCP, in the open phase the protocol establishes a connection and then the exchange of data takes place. Unreliable protocols don't need to establish a connection.

2.1 Execution Constructs

The controller accepts three types of execution constructs namely:

- serial
- parallel
- cluster

Execution Constructs

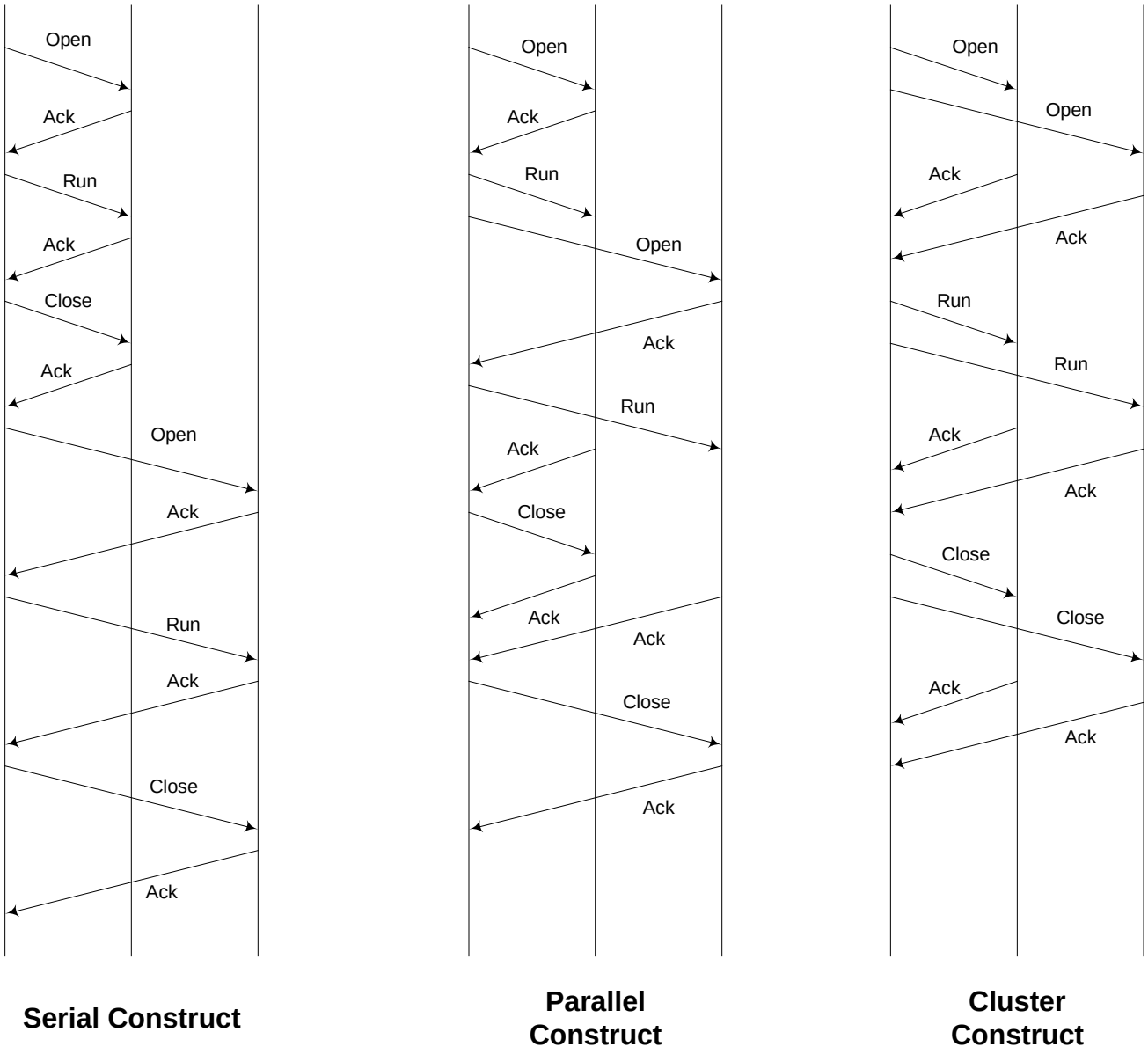

Figure 2 The Three types of Execution Constructs

When two daemons, Daemon A and Daemon B are inside a serial or parallel or cluster construct we observe the following behavior

serial - The open, run and close phases with respect to one daemon (say Daemon A) are finished before Daemon B is executed. So, Daemons A and B are executed serially.

parallel - When the run phase of Daemon A is in progress the open phase of Daemon B is initiated. The run phase of both the daemons is executed in parallel.

cluster - The open phase of Daemon A is executed and the run is executed only after the open phase of Daemon B is done. In this construct all the phases of both the daemons are done in parallel. This construct encloses a source and a sink.

3.0 Running NetSpec

To understand the different options with which NetSpec can be run we can just type netspec –h at the prompt to see NetSpec help.

```
newton [54] % netspec –h

        NetSpec User Interface, NetSpec Version 3.0 alpha 1

Usage: netspec <file> [options] <file>

If the file is not specified input is read from stdin

Options:

-s ipaddress   server to use (localhost)
-p port       which port to contact server on (42001)
-e file        error file output (stderr)
-d level      level of debug / error messages (0)
-i             Interactive mode, manual control
-m            pass it through m4
-c             pass it through cpp
-w            information about NetSpec
-h             this
```

The problem with multiple processes on multiple hosts is that debugging becomes a cumbersome and elaborate process. Debugging becomes easier if the daemon runs in standalone mode. Each of the daemons includes a full-fledged server that can run under inetd as well as standalone.

The –s option is used to run the daemon in the stand-alone mode and the –p option is used to specify the port on which the server is to be contacted. By typing the following command each daemon can be run in the standalone mode.

```
lovelace [5] % nstestd –s –p portnumber
```

To display the debug information and to trace the flow of control the –d option can be used. The –d option takes integer arguments and the argument represents the level of

verboseness. The –f option is used to facilitate running the daemon with the debugger. It instructs the daemon not to fork since this usually complicates matters within the debugger.

When we use the –i option we can run NetSpec in the interactive mode. When NetSpec is run in the interactive mode the following RCIP commands are displayed. This debugging mode is quite elaborate, with each command it is possible to specify parameters as appropriate for the daemon.

```
lovelace [5] % netspec tcp.single -i
NetSpec Version 3.0 alpha 1 Control Daemon lovelace[13459] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon lovelace [13457]
ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon marr[6311] ready

NetSpec debugging mode ready:
Type ? for help

rcips > ?

NetSpec RCIPS debugging mode description

Local commands are:

? or help
quit

RCIPS commands are structured as follows:
command; or command { parameter; parameter; .... }

rcips commands are :

setupCommand
openCommand
runCommand
finishCommand
closeCommand
teardownCommand
killCommand
resetCommand
paramsCommand
configCommand
reportCommand
For more details see the programmers manual (Not done yet)

rcips > setupCommand;
```

Each command can be entered by the user at the RCIPS prompt and executed. This option helps the user to step into each command and allows careful observation of the daemon under test.

3.1 Purpose of Scripts

All NetSpec experiments are based on the experiment script provided by the user. The script serves the following purposes:

- Identify the nodes involved in the experiment. A node is a generic term for a NetSpec daemon, which can be a control, test or measurement daemon.
- Identify the role assumed by each node
- Define the test parameters for each measurement/test daemon.
- Describe the relationship between nodes and thus describing the topology of the experiment.

4.0 NetSpec Daemons

Under the NetSpec framework, a host of daemons have been developed. These daemons help the user in running the tests on different environments. There is a Control Daemon, which essentially starts up the required test/measurement daemon according to the specification provided by the user in the script. The test/measurement daemon performs the test measures the required parameters and displays the measured information in the form of a report. A brief description of the different daemons under the NetSpec framework is dealt with in the forth-coming sections.

4.1 Test Daemon (nstestd)

For every connection in the experiment the control daemon creates a test daemon. The test daemons provide the capability to generate and sink a wide variety of TCP and UDP traffic. These test daemons concentrate on performing the traffic related tasks (send or receive data transferred across the connection). The test daemon is responsible for its own report generation after experiment execution is complete.

4.1.1 Description of user NetSpec Scripts and Examples

The user interface to NetSpec is a simple block structured language or otherwise a NetSpec script with keywords and user parameters. The appropriate NetSpec parsers will process this script and the output will be delivered to the lower levels of the software architecture.

The best way to learn how to create these scripts is by example. What follows is a variety of real scripts with the necessary explanation and suggestions on where they can be used.

- Full Stream Traffic Mode
- Burst Traffic Mode
- Queued Burst Traffic Mode
- Emulated Traffic types

4.1.2 Full Stream Traffic Scripts

In the full stream traffic mode, the hosts under test transmit data as fast as they can. This is the mode that ttcp supports. NetSpec on this mode can be used to estimate the maximum throughput that can be utilized in a specific communication channel, being either terrestrial or satellite. Table 10 given in Appendix explains all the key words used in the following scripts.

4.1.2.1 Full Stream Traffic: Script #1 - Point-to-Point TCP Traffic

Generally, the script given below is of the most widely type used for a TCP point-to-point connection. The first block specifies the characteristics of the traffic source, while the second block specifies the characteristics of the receiver host. The sender and receiver TCP systems used are machines in the local ATM network in the University of Kansas (KU). This script can be used in testing the performance in a WAN, as well as testing the performance over a satellite link, after the appropriate window size is specified for TCP using the bandwidth-delay product for achieving optimized performance.

```
1 cluster  {

2    test lovelace {
3       type = full (blocksize=32768, duration=30);
4       protocol = tcp (window=32768);
5       own = lovelace.atm:42020;
6       peer =marr.atm:42020;
7     }

8    test marr {

9       type = sink (blocksize=32768, duration=30);
10      protocol = tcp (window=32768);
11      own = marr.atm:42020;
12      peer = lovelace.atm:42020;
13    }

14  }
```

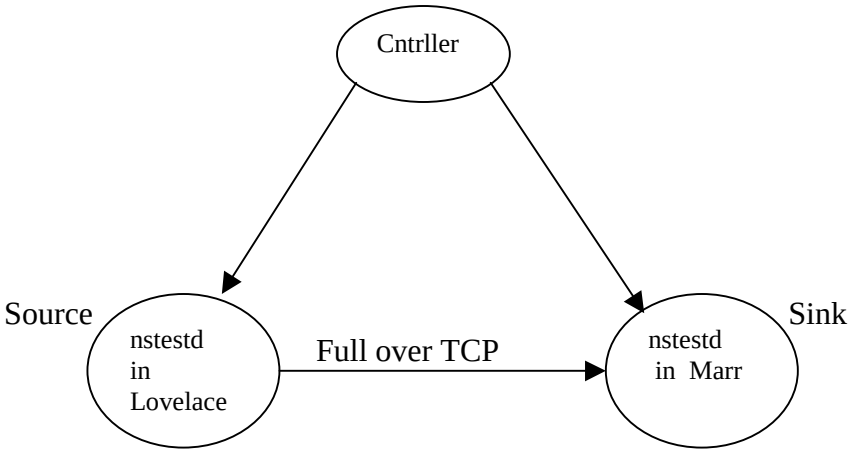

Figure 3 Test setup with the invoked daemons over TCP

In this particular example, the host with name lovelace is the sender system and the host with name marr is the receiver system. The sender (line 2) sends data in full stream (line 3) mode; it transmits 32768 bytes as fast as possible for 30 seconds (duration of experiment). For this data transfer TCP (line 4) is used in the transport layer with a window size of 32.7KB. All the keywords used in the script are described in Table 10 given in the appendix.

4.1.2.2 Report Generated

This is the report generated when the script given above was run.

```
lovelace [5] % netspec tcp.single
NetSpec Version 3.0 alpha 1 Control Daemon lovelace[16096] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon lovelace[16075]
ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon marr[711] ready

NetSpec Control daemon, NetSpec Version 3.0 alpha 1

Report from lovelace[16096] generated @ 10/08/98 17:43:44 CDT


NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from lovelace[16075]

  Start                         : 10/08/98 17:43:06 CDT
  End                           : 10/08/98 17:43:36 CDT
  Duration                      :          30.003 sec

Test data
```

```
  Own address                  : lovelace.atm.ittc.ukans.edu:42020
  Peer address                 : marr.atm.ittc.ukans.edu:42020
  Type                         :            full
  Blocks transmitted           :           11440 blocks
  Segments transmitted         :           11440 segments
  Bytes transmitted            :       374865920 bytes
  Thruput transmitted          :          99.955 Mbps
 Blocksize
    Minimum                    :           32768 bytes
    Maximum                    :           32768 bytes
    Mean                       :       32768.000 bytes
    Variance                   :           0.000 bytes
    Deviation                  :           0.000 bytes
  SegmentSize
    Minimum                    :           32768 bytes
    Maximum                    :           32768 bytes
    Mean                       :       32768.000 bytes
    Variance                   :           0.000 bytes
    Deviation                  :           0.000 bytes

Protocol data (socket options)

  Protocol                     :          TCP/IP
  Socket type                  :          STREAM
  Transmit
    buffer (window)            :           36560 bytes
    low water                  :            9140 bytes
    time out                   :               0 usec
    copy avoid                 :           False
  Receive
    buffer (window)            :           36560 bytes
    low water                  :               1 bytes
    time out                   :               0 usec
    copy avoid                 :           False
  TCP segment size             :            9140 bytes
  Debug                        :           False
  No routing                   :           False
  Loopback                     :           False
  No Delay                     :           False
  Server                       :            True

Process stats (rusage)

  System CPU time              :        11465072 us
  User CPU time                :          269376 us
  Total CPU time               :        11734448 us
  Page faults                  :               0
  Page reclaims                :               3
  Swaps                        :               0
  Block inputs                 :               0
  Block outputs                :               0
  Message inputs               :               0
  Message outputs              :           11441
  Signals                      :               1
  Voluntary context switches   :           20138
  Forced context switches      :             256
```

```
NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from marr[711]

  Start                         : 10/08/98 17:31:43 CDT
  End                           : 10/08/98 17:32:14 CDT
  Duration                      :          30.281 sec

Test data

  Own address                   : marr.atm.ittc.ukans.edu:42020
  Peer address                  : lovelace.atm.ittc.ukans.edu:42020
  Type                          :            sink
  Segments received             :           11440 segments
  Bytes received                :       374865920 bytes
  Thruput received              :          99.038 Mbps
  Segmentsize
    Minimum                     :           32768 bytes
    Maximum                     :           32768 bytes
    Mean                        :       32768.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes

Protocol data (socket options)

  Protocol                      :          TCP/IP
  Socket type                   :          STREAM
  Transmit
    buffer (window)             :           36560 bytes
    low water                   :            9140 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer (window)             :           36560 bytes
    low water                   :               8 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  TCP segment size              :            9140 bytes
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False
  No Delay                      :           False
  Server                        :           False

Process stats (rusage)

  System CPU time               :         6819312 us
  User CPU time                 :          407968 us
  Total CPU time                :         7227280 us
  Page faults                   :               0
  Page reclaims                 :               2
  Swaps                         :               0
  Block inputs                  :               0
  Block outputs                 :               0
  Message inputs                :           62996
  Message outputs               :               0
```

```
Signals                         :              1
Voluntary context switches      :          39658
Forced context switches         :           1530
```

The report gives the statistics of the test in term of the blocks and bytes transmitted and received. Also the throughput which is a measure of the amount of data a communications channel can carry is measured from both the sender's and the receiver's side. It helps the user to find out the state of the network between two endpoints. Helps the user evaluate the protocol used for transmission.

4.1.2.3 Full Stream Traffic: Script #2 - Point-to-Point UDP Traffic

This script can be used for performance evaluation of a point-to-point LAN and WAN communication link (terrestrial or satellite) where the UDP protocol is used instead. The block structure is the same as in script #1. The only difference is that the UDP protocol is specified instead of TCP (line 4 and 10), and no window size is needed; UDP just sends IP packets without scaling windows or retransmission schemes.

```
1  cluster {

2       test marr {

3               type = full (blocksize=8192, duration=30);
4                protocol = udp;
5               own = marr.atm:42005;
6               peer = lovelace.atm:42005;

7       }

8       test lovelace {

9               type = sink (blocksize=8192, duration=30);
10              protocol = udp;
11               own = lovelace.atm:42005;
12               peer =marr.atm:42005;

13      }

14 }
```

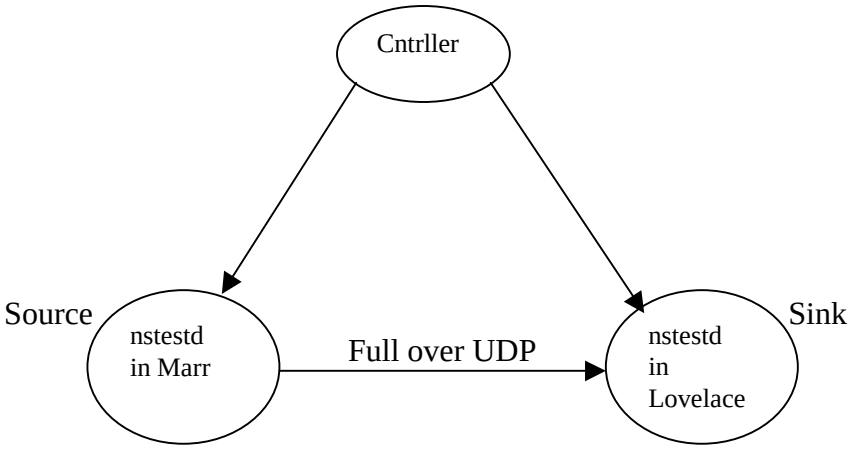

Figure 4 Test Setup over UDP

In this particular example, the host with name lovelace is the sender system and the host with name marr is the receiver system. The sender sends data in full stream mode (line 3) it transmits 8192 bytes as fast as possible for 30 seconds (duration of experiment). For this data transfer UDP is used in the transport layer.

4.1.2.4 Report Generated

Unlike TCP, UDP does not ensure data is safely received at the other end. If cells are dropped because of traffic congestion, other cells in the same UDP packet are discarded due to incomplete error at the receiving host. For UDP experiments, in post processing the loss of UDP packets is evaluated. By correlating the records of UDP segments using the sequence number, we can easily derive the UDP segment losses.

Segment Losses (%) = Number of missing segments / Number of Txd segments * 100%

From this we can see the difference between TCP and UDP since TCP is connection oriented and that ensures that all the data transmitted is received. By comparing the reports generated by using TCP and UDP we are able to see practically, the difference between a TCP and a UDP connection.

```
hopper [12] % netspec udp.single
NetSpec Version 3.0 alpha 1 Control Daemon hopper[1065] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon marr[2267] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon lovelace[18482]
ready

NetSpec Control daemon, NetSpec Version 3.0 alpha 1

Report from hopper[1065] generated @ 10/09/98 15:10:40 CDT
```

```
NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from marr[2267]

  Start                         : 10/09/98 15:10:08 CDT
  End                           : 10/09/98 15:10:38 CDT
  Duration                      :          30.001 sec

Test data

  Own address                   : marr.atm.ittc.ukans.edu:42005
  Peer address                  : lovelace.atm.ittc.ukans.edu:42006
  Type                          :            full
  Blocks transmitted            :           59479 blocks
  Segments transmitted          :           59479 segments
  Bytes transmitted             :       487251968 bytes
  Thruput transmitted           :         129.929 Mbps
  Blocksize
    Minimum                     :            8192 bytes
    Maximum                     :            8192 bytes
    Mean                        :        8192.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes
  SegmentSize
    Minimum                     :            8192 bytes
    Maximum                     :            8192 bytes
    Mean                        :        8192.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes

Protocol data (socket options)

  Protocol                      :          UDP/IP
  Socket type                   :          DGRAM
  Transmit
    buffer                      :            9216 bytes
    low water                   :            4608 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer                      :           41600 bytes
    low water                   :               1 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False

Process stats (rusage)

  System CPU time               :        14535568 us
  User CPU time                 :          634400 us
  Total CPU time                :        15169968 us
  Page faults                   :               0
  Page reclaims                 :               0
  Swaps                         :               0
```

```
  Block inputs                  :              0
  Block outputs                 :              0
  Message inputs                :              0
  Message outputs               :          60995
  Signals                       :              1
  Voluntary context switches    :              0
  Forced context switches       :           3370


NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from lovelace[18482]

  Start                         : 10/09/98 15:21:32 CDT
  End                           : 10/09/98 15:22:02 CDT
  Duration                      :          30.031 sec

Test data

  Own address                   : lovelace.atm.ittc.ukans.edu:42006
  Peer address                  : marr.atm.ittc.ukans.edu:42005
  Type                          :            sink
  Segments received             :           59144 segments
  Bytes received                :       484507648 bytes
  Thruput received              :         129.068 Mbps
  Segmentsize
    Minimum                     :            8192 bytes
    Maximum                     :            8192 bytes
    Mean                        :         8192.000 bytes
    Variance                    :            0.000 bytes
    Deviation                   :            0.000 bytes

Protocol data (socket options)

  Protocol                      :          UDP/IP
  Socket type                   :          DGRAM
  Transmit
    buffer                      :           65496 bytes
    low water                   :            8192 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer                      :           65496 bytes
    low water                   :               1 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False

Process stats (rusage)

  System CPU time               :         6662176 us
  User CPU time                 :          556320 us
  Total CPU time                :         7218496 us
  Page faults                   :               0
```

```
  Page reclaims                 :              1
  Swaps                         :              0
  Block inputs                  :              0
  Block outputs                 :              0
  Message inputs                :          59144
  Message outputs               :              0
  Signals                       :              1
  Voluntary context switches    :          58186
  Forced context switches       :            761
```

4.1.2.5 Full Stream Traffic: Script #3 - Parallel Connections

This script describes how the parallel construct works. An example where this script could be used is to emulate multicasting. When a receiver has registered in different groups to receive packets we can emulate that scenario in the parallel construct with multiple transmitters(in this case three TCP sources) transmitting to a single receiver.

```
parallel {

cluster {

test wiley {
        type = full (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
        own = wiley.atm:45001;
        peer = faraday.atm:45000;
}

test faraday {
        type = sink (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45000;
        peer = wiley.atm:45001;
}
}

cluster {

test galaga {
        type = full (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
        own = galaga.atm:45002;
        peer = faraday.atm:45003;
}

test faraday {
        type = sink (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
```

```
        own = faraday.atm:45003;
        peer = galaga.atm:45002;
}
}

cluster {

test elmer {
        type = full (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
        own = elmer.atm:45004;
        peer = faraday.atm:45005;
}

test faraday {
        type = sink (blocksize=131072, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45005;
        peer = elmer.atm:45004;
}
}
}
```

This script carries out a performance evaluation test of three TCP parallel connections. The above script was used to evaluate the performance of TCP/IP hosts on ATM networks over an OC-3c satellite link, under congestion conditions. The same script can be modified to be used with UDP [just change TCP to UDP and delete the window = parameter], or with serial connections [just change parallel to serial]. It can support an arbitrary number of parallel or serial connections (point-to-multipoint, multipoint-to-point) and can be used on any network architecture or size.

In this particular example, the hosts with names wiley, elmer and galaga are the sender systems and the host with alias name faraday is the receiver system. The senders sends data in full stream mode; each one transmits 131072 bytes as fast as possible for 10 minutes (duration of experiment). For this data transfer TCP is used with a window size of 10 MB. As you can observe, through our script we can change the parameters accordingly to the adopted testbed we want to test. Since in this experiment we were testing a satellite link, we used a big TCP window size (10MB) and we ran the experiment for a longer period of time (600 seconds).

4.1.3 Burst Traffic scripts

In the burst traffic mode, the hosts under test transmit data at specific intervals, specified by the burst period as shown in the scripts below. The user passes the burst size and the burst period as parameters. This mode is very useful in real world experiments where rate mismatches might reduce the throughput dramatically. An example of this is TCP over ATM architectures. It has been proven that rate mismatch reduces throughput, and that's why traffic shaping must be enforced on the source end systems to boost performance. Therefore, running the appropriate scripts on this mode, one can shape the source traffic to a desirable rate which matches the destination or the link rate. This mode is tested over LANs, WANs, and satellite environments with very good results.

Table 10 given in Appendix explains all the key words used in the following scripts.

4.1.3.1 Burst Traffic: Script #1
Point-to-Point Traffic shaped at 100 Mbps

```
cluster {

  test lovelace {
        type = burst (blocksize=125000, period = 10000,  duration=10);
        protocol = tcp (window=1048576);
        own =  129.237.166.8:42020;
        peer = 129.237.166.7:42020;
  }

  test marr {

        type = sink (blocksize=125000, duration=10);
        protocol = tcp (window=1048576);
        own = 129.237.166.7:42020;
        peer = 129.237.166.8:42020;
  }
}
```

The first block specifies the characteristics of the traffic source, while the second block specifies the characteristics of the receiver host. The sender and receiver TCP systems used are machines in the local ATM network in the University of Kansas (KU). This script can be used in testing the performance in a LAN and a WAN, as well as testing the performance over a satellite link, where traffic shaping is necessary due to link mismatches. The traffic in this script is shaped to 100 Mbps. This is calculated by:

$$Throughput = (blocksize * 8)/(period * 10^{(-3)}) \ [b/s]$$

Where blocksize and period are the values used in the script.

4.1.3.2 Report Generated

```
hopper [2] % netspec burst.atm
NetSpec Version 3.0 alpha 1 Control Daemon hopper[1211] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon lovelace[19696]
ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon marr[2426] ready

NetSpec Control daemon, NetSpec Version 3.0 alpha 1

Report from hopper[1211] generated @ 10/09/98 17:14:15 CDT


NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from lovelace[19696]

  Start                         : 10/09/98 17:25:23 CDT
  End                           : 10/09/98 17:25:33 CDT
  Duration                      :          10.022 sec

Test data

  Own address                   : lovelace.atm.ittc.ukans.edu:42020
  Peer address                  : marr.atm.ittc.ukans.edu:42020
  Type                          :           burst
  Blocks transmitted            :            1001 blocks
  Segments transmitted          :            1001 segments
  Bytes transmitted             :       125125000 bytes
  Thruput transmitted           :          99.879 Mbps
  Failed cycles                 :               0
  Blocksize
    Minimum                     :          125000 bytes
    Maximum                     :          125000 bytes
    Mean                        :      125000.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes
  SegmentSize
    Minimum                     :          125000 bytes
    Maximum                     :          125000 bytes
    Mean                        :      125000.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes
  Repeat count
    Minimum                     :               1
    Maximum                     :               1
    Mean                        :           1.000
    Variance                    :           0.000
    Deviation                   :           0.000
  Period
    Minimum                     :           10000 usec
    Maximum                     :           10000 usec
    Mean                        :       10000.000 usec
    Variance                    :           0.000 usec
    Deviation                   :           0.000 usec
```

```
Protocol data (socket options)

  Protocol                      :          TCP/IP
  Socket type                   :          STREAM
  Transmit
    buffer (window)             :         1051100 bytes
    low water                   :            9140 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer (window)             :         1051100 bytes
    low water                   :               1 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  TCP segment size              :            9140 bytes
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False
  No Delay                      :           False
  Server                        :            True

Process stats (rusage)

  System CPU time               :         1325408 us
  User CPU time                 :           79056 us
  Total CPU time                :         1404464 us
  Page faults                   :               0
  Page reclaims                 :              14
  Swaps                         :               0
  Block inputs                  :               0
  Block outputs                 :               0
  Message inputs                :               0
  Message outputs               :            1001
  Signals                       :            1001
  Voluntary context switches    :            1001
  Forced context switches       :             495



NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from marr[2426]

  Start                         : 10/09/98 17:13:59 CDT
  End                           : 10/09/98 17:14:09 CDT
  Duration                      :          10.129 sec

Test data

  Own address                   : marr.atm.ittc.ukans.edu:42020
  Peer address                  : lovelace.atm.ittc.ukans.edu:42020
  Type                          :            sink
  Segments received             :            1001 segments
  Bytes received                :       125125000 bytes
  Thruput received              :          98.824 Mbps
  Segmentsize
```

```
    Minimum                     :          125000 bytes
    Maximum                     :          125000 bytes
    Mean                        :       125000.000 bytes
    Variance                    :            0.000 bytes
    Deviation                   :            0.000 bytes

Protocol data (socket options)

  Protocol                      :          TCP/IP
  Socket type                   :          STREAM
  Transmit
    buffer (window)             :           36560 bytes
    low water                   :            9140 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer (window)             :           36560 bytes
    low water                   :               8 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  TCP segment size              :            9140 bytes
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False
  No Delay                      :           False
  Server                        :           False

Process stats (rusage)

  System CPU time               :         1902224 us
  User CPU time                 :          103456 us
  Total CPU time                :         2005680 us
  Page faults                   :               0
  Page reclaims                 :              14
  Swaps                         :               0
  Block inputs                  :               0
  Block outputs                 :               0
  Message inputs                :           15073
  Message outputs               :               0
  Signals                       :               1
  Voluntary context switches    :           12794
  Forced context switches       :             720
```

4.1.3.3 Burst Traffic: Script #2
Parallel Connections: Traffic Shaping at 30 Mbps per host

```
parallel {

cluster {

test wiley {
        type = burst (blocksize=37500,period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = wiley.atm:45000;
        peer = faraday.atm:45001;
}

test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45000;
        peer = wiley.atm:45001;
}
}

cluster {

test galaga {
        type = burst (blocksize=37500, period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = galaga.atm:45002;
        peer = faraday.atm:45003;
}
test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45003;
        peer = galaga.atm:45002;
}
}

cluster {

test elmer {
        type = burst (blocksize=37500, period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = elmer.atm:45004;
        peer = faraday.atm:45005;
```

```
}

test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45005;
        peer = elmer.atm:45004;
}
}
}
```

This script carries out a performance evaluation test of three TCP parallel connections where traffic shaping is performed. It is a good example of multi-casting, where three TCP sources transmit data to one receiver simultaneously, at a rate of

(37500 * 8) bits/10 ms = 30 Mbps

each TCP source host [This comes out to an aggregate paced throughput of 90 Mbps].

The above script was used to evaluate the performance of TCP/IP hosts on ATM networks over an OC-3c satellite link. Since the OC-3 satellite link would not be able to forward cells coming from the three OC-3c TCP over ATM connections as fast as the incoming rate, congestion will take place, cells will be dropped at the ATM switches, TCP retransmissions will take place and thus throughput will degrade. Therefore using NetSpec in burst mode and pacing the traffic to an appropriate aggregate rate, one can avoid overflowing the link and thus keeping performance to a decent level. The same script can be modified to be used with UDP [just change TCP to UDP and delete the window = parameter], or with serial connections [just change parallel to serial]. It can support an arbitrary number of parallel or serial connections (point-to-multipoint, multipoint-to-point) and can be used on any network architecture or size.

4.1.4 Queued Burst Traffic scripts

In the queued burst traffic mode, the hosts under test transmit data at specific intervals, specified by the burst period as shown in the scripts below. This mode is a variation of the basic burst algorithm and the burst size and burst period are passed as parameters by the user. The advantage of this algorithm is that variations in available line rate will not cause it to miss blocks generated by interrupts arriving before previous write completes. The drawback is that traffic characteristics are influenced by the queuing delay. The usefulness of this mode is similar to that of the burst mode, since basically they achieve application level traffic shaping, which is necessary where rate mismatches are present, that will reduce the throughput dramatically. An example of this is TCP over ATM architectures. It has been proved that rate mismatching reduces throughput, and that's why traffic shaping must be enforced on the source end systems to boost performance. Therefore, running the appropriate scripts on this mode, one can shape the source traffic to a desirable rate which matches the destination or the link rate. This mode is tested over LANs, WANs, and satellite environments with incredible results.

4.1.4.1 Queued Burst Traffic: Script #1 : Point-to-Point Traffic shaped at 100 Mbps

```
cluster {

  test lovelace {
        type = burstq (blocksize=125000, period =10000, duration=10);
        protocol = tcp (window=1048576);
        own = lovelace.atm:42015;
        peer = marr.atm:42015;
  }

  test marr {
        type = sink (blocksize=125000, duration=10);
        protocol = tcp (window=1048576);
        own = marr.atm:42015;
        peer = lovelace.atm:42015;
  }
}
```

The first block specifies the characteristics of the traffic source, while the second block specifies the characteristics of the receiver host. The sender and receiver TCP systems used are machines in the local ATM network in the University of Kansas (KU). This script can be used in testing the performance in a LAN and a WAN, as well as testing the performance over a satellite link, where traffic shaping is necessary due to link mismatches. The traffic in this script is shaped to 100 Mbps. This is calculated by:

$$Throughput = (blocksize * 8)/(period * 10^{(-3)}) [b/s]$$

Where blocksize and period are the values used in the script.

4.1.4.2 Report Generated

```
hopper [13] % netspec burstq.atm
NetSpec Version 3.0 alpha 1 Control Daemon hopper[1089] ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon lovelace[19127]
ready
NetSpec Version 3.0 TCPgamma 1, UDPalpha 1 Test Daemon marr[2348] ready

NetSpec Control daemon, NetSpec Version 3.0 alpha 1

Report from hopper[1089] generated @ 10/09/98 15:42:23 CDT

NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from lovelace[19127]

  Start                         : 10/09/98 15:53:31 CDT
  End                           : 10/09/98 15:53:41 CDT
  Duration                      :          10.663 sec

Test data

  Own address                   : lovelace.atm.ittc.ukans.edu:42015
  Peer address                  : marr.atm.ittc.ukans.edu:42015
  Type                          :          burstQ
  Blocks transmitted            :            1000 blocks
  Segments transmitted          :            1000 segments
  Bytes transmitted             :       125000000 bytes
  Thruput transmitted           :          93.689 Mbps
  Leftover blocks               :               0
  Dropped blocks                :               0
  Blocksize
    Minimum                     :          125000 bytes
    Maximum                     :          125000 bytes
    Mean                        :      125000.000 bytes
    Variance                    :           0.000 bytes
    Deviation                   :           0.000 bytes
  SegmentSize
    Minimum                     :               0 bytes
    Maximum                     :          125000 bytes
    Mean                        :      124875.000 bytes
    Variance                    :    15625000.000 bytes
    Deviation                   :        3952.847 bytes
  Repeat count
    Minimum                     :               1
    Maximum                     :               1
    Mean                        :           1.000
    Variance                    :           0.000
    Deviation                   :           0.000
  Period
    Minimum                     :           10000 usec
```

```
    Maximum                     :           10000 usec
    Mean                        :       10000.000 usec
    Variance                    :           0.000 usec
    Deviation                   :           0.000 usec
  Queue
    Minimum                     :               0 blocks
    Maximum                     :               3 blocks
    Mean                        :           0.095 blocks
    Variance                    :           0.118 blocks
    Deviation                   :           0.344 blocks

Protocol data (socket options)

  Protocol                      :          TCP/IP
  Socket type                   :          STREAM
  Transmit
    buffer (window)             :         1051100 bytes
    low water                   :            9140 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  Receive
    buffer (window)             :         1051100 bytes
    low water                   :               1 bytes
    time out                    :               0 usec
    copy avoid                  :           False
  TCP segment size              :            9140 bytes
  Debug                         :           False
  No routing                    :           False
  Loopback                      :           False
  No Delay                      :           False
  Server                        :            True

Process stats (rusage)

  System CPU time               :         1979328 us
  User CPU time                 :           73200 us
  Total CPU time                :         2052528 us
  Page faults                   :               0
  Page reclaims                 :              14
  Swaps                         :               0
  Block inputs                  :               0
  Block outputs                 :               0
  Message inputs                :               0
  Message outputs               :             999
  Signals                       :            1000
  Voluntary context switches    :             918
  Forced context switches       :            1098



NetSpec Test daemon, NetSpec Version 3.0 TCPgamma 1, UDPalpha 1

Report from marr[2348]

  Start                         : 10/09/98 15:42:07 CDT
  End                           : 10/09/98 15:42:18 CDT
  Duration                      :           10.758 sec
```

```
Test data

  Own address                  : marr.atm.ittc.ukans.edu:42015
  Peer address                 : lovelace.atm.ittc.ukans.edu:42015
  Type                         :            sink
  Segments received            :             999 segments
  Bytes received               :       124875000 bytes
  Thruput received             :          92.864 Mbps
  Segmentsize
    Minimum                    :          125000 bytes
    Maximum                    :          125000 bytes
    Mean                       :       125000.000 bytes
    Variance                   :            0.000 bytes
    Deviation                  :            0.000 bytes

Protocol data (socket options)

  Protocol                     :          TCP/IP
  Socket type                  :          STREAM
  Transmit
    buffer (window)            :           36560 bytes
    low water                  :            9140 bytes
    time out                   :               0 usec
    copy avoid                 :           False
  Receive
    buffer (window)            :           36560 bytes
    low water                  :               8 bytes
    time out                   :               0 usec
    copy avoid                 :           False
  TCP segment size             :            9140 bytes
  Debug                        :           False
  No routing                   :           False
  Loopback                     :           False
  No Delay                     :           False
  Server                       :           False

Process stats (rusage)

  System CPU time              :         2419504 us
  User CPU time                :          101504 us
  Total CPU time               :         2521008 us
  Page faults                  :               0
  Page reclaims                :              14
  Swaps                        :               0
  Block inputs                 :               0
  Block outputs                :               0
  Message inputs               :           15186
  Message outputs              :               0
  Signals                      :               1
  Voluntary context switches   :           12821
  Forced context switches      :             718
```

4.1.4.3 Queued Burst Traffic: Script #2
Parallel Connections: Traffic Shaping at 30 Mbps per host

```
parallel {

cluster {

test wiley {
        type = burstq (blocksize=37500,period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = wiley.atm:45000;
        peer = faraday.atm:45001;
}

test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45000;
        peer = wiley.atm:45001;
}
}

cluster {

test galaga {
        type = burstq (blocksize=37500, period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = galaga.atm:45002;
        peer = faraday.atm:45003;
}

test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45003;
        peer = galaga.atm:45002;
}
}

cluster {

test elmer {
        type = burstq (blocksize=37500, period=10000, duration=600);
        protocol = tcp (window=10485760);
        own = elmer.atm:45004;
        peer = faraday.atm:45005;
```

```
}

test faraday {
        type = sink (blocksize=37500, duration=600);
        protocol = tcp (window=10485760);
        own = faraday.atm:45005;
        peer = elmer.atm:45004;
}
}
}
```

Similar to the other parallel scripts, this one is also a good example of multicasting, where three TCP sources transmit data to one receiver simultaneously, at a rate of

(37500 * 8) bits/10ms = 30 Mbps

each TCP source host [This comes out to an aggregate paced throughput of 90 Mbps].

The above script was used to evaluate the performance of TCP/IP hosts on ATM networks over an OC-3c satellite link.

4.1.5 Emulated Traffic types and scripts

NetSpec can produce many types of traffic flows[2] where the user can specify the statistical properties of each flow. Thus NetSpec can generate different emulated types of traffic. In this section, we will explain scripts with all the emulated traffic supported by NetSpec.

Emulated Traffic Models

- TELNET Traffic
- FTP Traffic
- VBR Traffic (Video-Teleconference, MPEG)
- CBR Traffic (Voice)
- WWW Traffic

4.1.5.1 TELNET Traffic

Four basic parameters characterize TELNET traffic:

- TELNET Session Interarrival Time (seconds); Modeled as a Poisson process with fix hourly rates.
- TELNET Session Duration (seconds); Modeled as a log-normal distribution.
- Packet Interarrival Time (seconds); Modeled as a Pareto distribution.
- Packet size (bytes); Follows a special distribution.

All of the above parameters have been implemented in NetSpec to deliver TELNET traffic.

Example Script

Table 11 given in Appendix explains all the key words used in the following scripts.

```
cluster {

test galaga {
        type = session (type = burstq (blocksize=telnetPacketSize,
                                       period=telnetPacketInterarrival,
                                       buffer=262144,
                                      duration=telnetSessionDuration(mean=7.91,
                                      stdDeviation=2.96)),
                               interarrival=telnetSessionInterarrival(lambda=0.0000001),
                                       duration= 900);
        protocol = tcp (window= 262144);
```

```
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
}

test hopper {
        type = sink (buffer=262144, duration=900);
        protocol = tcp (window=262144, rcvlowat=8);
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

The blocksize and the period are using the fixed models, telnetPacketSize and telnetPacketInterarrival. Duration of each session is set to be the telnetSessionDuration distribution with mean = 7.91 and standard deviation = 2.96. This means that the mean of telnet sessions is 2^7.91; approximately 240 seconds. The interarrival of telnet session has a mean of 1/0.0000001 microseconds = 10 seconds. The duration of the test is 900 seconds = 15 minutes.

4.1.5.2 FTP Traffic

Compared to TELNET traffic, FTP traffic doesn't have the duration and item interarrival time setups. This is because the duration of each FTP session is dependent upon the network capacity, such as link rate. The faster the link rate, the more items the network can transfer in the same amount of time.

The following three parameters are needed to characterize FTP traffic:

- FTP Session Interarrival Time (seconds); Modeled as a Poisson process within one hour intervals.
- FTP number of items transferred; Follows a special distribution.
- FTP item size (bytes); Modeled as log-normal distribution.

Example Script

```
cluster {

test galaga {
        type = burstq (blocksize=ftpItemSize,
                        repeats=ftpNOfItems,
                        period=ftpSessionInterarrival(lambda=0.00001),
                        buffer=262144,
                        duration= 1800);
        protocol = tcp (window= 262144);
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
```

```
}

test hopper {
        type = sink (blocksize=262144, duration=900);
        protocol = tcp (window=262144)
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

ftpItemSize and ftpNOfItems are the fixed models. In this script, the interarrival time of ftp sessions has a mean of 1/0.00001 = 100 milliseconds. The duration of the whole test is about half an hour (1800 seconds).

4.1.5.3 VBR Traffic (e.g. Video)

Broadband integrated networks are expected to carry substantial portions of the video services. Accurate source modeling of video and other services is essential to develop a network that achieves pre-defined quality of services and cost-efficiency.

Generally, there are two types of video traffic:

(1) Teleconference video stream, and
(2) MPEG video stream.

Each of these video streams has difference characteristics based on its nature of object motions and use of compression algorithms. NetSpec is capable of producing both video traffic streams as specified.

1.Video Teleconference

Two basic parameters characterize Video teleconference traffic.

- Frame Interarrival Time(seconds); [1/frame rate].
  It's a constant value: In NTSC systems is 33 ms (30 frames/sec). In PAL systems is 40 ms (25 frames/sec).If using 25 to 30 frames/sec rates, it will produce high quality video stream that requires a large portion of network bandwidth. Typical conference calls with high compression technique that produce acceptable quality often only require 5 to 15 frames/sec rates. Generally, 12 frame/sec (83 milliseconds/frame) is 0 commonly used.

- Number of Cells per frame. Modeled as a gamma distribution.

Both of the above parameters have been implemented in NetSpec to deliver Video Teleconference traffic.

Example Script

```
cluster {

test galaga {
        type = burstq (blocksize=videoTeleConferenceFrameSize(scale=42.50,
        shape=3),
                        period=83000,
                        duration= 600);
        protocol = tcp (window= 262144);
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
}

test hopper {
        type = sink (buffer=131072, duration=900);
        protocol = tcp (window=262144, rcvlowat=8);
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

The constant period is 83000 microseconds (83 ms); approximately 12 frames/sec [$1/(83000*10^{(-6)})$].

2.MPEG (Motion Picture Experts Group) Video stream

The basic parameters needed to characterize MPEG Video traffic are:

- Frame Interarrival Time (seconds);
  For high quality motion pictures, a high frame rate is often required. The typical constant interframe period is at least 33 ms (30 frames/sec) for NTSC standard systems and 40 ms (25 frames/sec) for PAL standard systems.

- Scene length;
  A video stream consists of several segments such that the sizes of I frames in each segment are close in value. The length of a scene (in I frames) is modeled as a geometric distribution.

- Number of Cell per frame;
  There are three types of coded frames : Intra-coded (I); Prediction (P), and bidirectional (B) MPEG frames. The sizes of all three coded frames are modeled as log-normal distributions with different mean and standard deviation.

Both of the above parameters have been implemented in Netspec to deliver MPEG Video traffic.

Example Script

```
cluster {

test galaga {
        type = burstq (blocksize=videoMPEGFrameSize(sceneLengthMean=10.5,
                                                    Imean=5.1968,
                                                    IstdDeviation=0.2016,
                                                    Pmean=3.7380,
                                                    PstdDeviation=0.5961,
                                                    Bmean=2.8687,
                                                    BstdDeviation=0.2675,
                                                    period=33000,
                                                    duration= 300);
        protocol = tcp (window= 262144);
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
}

test hopper {
        type = sink (blocksize=131072, duration=300);
        protocol = tcp (window=262144, rcvlowat=8);
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

The interarrival time of frames is 33000 microseconds; approximately 30 frames/sec.

4.1.5.4 CBR Traffic (e.g. Voice)

The standard way for digitizing the voice signal is to bandlimit the signal to a band from 100Hz to 3400Hz and then to sample that at 8kHz. Each sample is given 8 bits. This produces the standard bit rate of 64kb/s for voice.

The following parameters are needed to characterize Voice traffic.

- Voice Session Interarrival Time (seconds);
  The connection arrivals can be modeled by a Poisson process with fixed hourly rates within one-hour periods.

- Voice Session Duration (holding time);
  Modeled as an exponential distribution.

• CBR Voice Packet Interarrival Time;
  This type of traffic can be simply described by its peak rate. For a telephone speech, 64kbits/sec is a standard CBR rate. The shorter the packet interarrival time, the better quality of voice. An ATM cell has a 48-byte load. Based on common voice 8000 Hz sampling rate, it takes 6 milliseconds to transmit a ATM cell.

• Voice Packet Size;
  The packet size is determined by the data transfer rate. For a 64kbits/sec telephone voice stream, the packet size is 144 bytes with a 18 msec interarrival rate.

All of the above parameters have been implemented in Netspec to deliver Voice and other CBR traffic[3].

Example Script

```
cluster {

test galaga {
        type = session (type = burstq (blocksize=144,
                                       period=18000,
                                   duration=voiceSessionDuration(lambda=0.004167),
                        interarrival=voiceSessionInterarrival(lambda=0.0000001),
                                       duration= 1800);
        protocol = tcp (window= 65536);
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
}

test hopper {
        type = sink (blocksize=144, duration=1800);
        protocol = tcp (window=65536, rcvlowat=8);
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

The blocksize is 144 bytes and the constant period is 18 miliseconds. As a result, the rate is 144*8 bits/18 miliseconds = 64 kbits/sec. The mean of the voice sessions is about 1/0.004167 seconds = 3 minutes.

4.1.5.5 World Wide Web Traffic

The traffic of World Wide Web (WWW) has increased exponentially due to the explosion of the information superhighway and includes a significant portion of traffic that is generated by the WWW browsers.

The parameters needed to characterize WWW traffic are:

• Mean Inter-Request Time;
  Modeled as a Poisson process with a fixed rate.
• Document Transfer Size;
  Modeled as a Pareto Distribution.

The above parameters have been implemented in Netspec to deliver WWW traffic.

Example Script

```
cluster {

test galaga {
        type = burstq (blocksize=WWWItemSize,
                        period=WWWRequestInterarrival(lambda=0.000002),
                        buffer=262144,
                       duration= 1200);
        protocol = tcp (window= 262144);
        own = galaga.atm:45000;
        peer = hopper.atm:45001;
}



test hopper {
        type = sink (buffer=262144, duration=1200);
        protocol = tcp (window=262144, rcvlowat=8);
        own = hopper.atm:45000;
        peer = galaga.atm:45001;
}
}
```

The Request Interarrival Time is set with a lambda = 0.000002. This corresponds to a mean of 1/0.000002 = 500 milliseconds.

4.2 Data Stream Daemon (nsdstrd)

The NetSpec Test Daemon (nstestd) generates various types of traffic and delay variations and measures the end-to-end performance in terms of throughput. While the Test Daemon also provides a way to collect time stamps of individual packets at the application level, it is often desirable to collect the event traces from the lower levels of the system.

Data Stream Driver Interface (DSKI) [http://hegel.ittc.ukans.edu/projects/datastream/index.html] is a tool to collect event traces and data from the Operating System kernel. To allow data collection from the Operating System in the NetSpec framework, the nsdstrd was written. The nsdstrd thus functions as a measurement daemon. The Data Stream daemon allows the collection of event traces and counter values from the OS kernel. The kernel must be instrumented with tap points as described in the DSKI document. The DSKI is only supported in the Digital UNIX and Linux. To make use of the Data Stream daemon, you must have access to the OS source code.

4.2.1 Features

1. Event Traces (Active Source)

- Collect event traces for a subset of events from the kernel.
- Can enable all events in a family.
- Can specify the length of experiment by the duration (in second) or by the number of events.
- Can specify one special parameter to filter trace.

2. Counter

- Collect counter values for a subset of counters from the kernel.
- Can enable all counter in a family.
- Collection is done periodically for a specified duration.

4.2.2 NetSpec Script

1. Active Source

```
dstream {
        type = active (numevents = x, duration = y, param = z);
        family1 = set ( eventnames ) | all;
        family2 = set ( eventnames ) | all;
}
```

Parameters:

- numevents: The maximum number of events to collect.
- duration: The length of experiment in seconds. If duration is not specified, the experiment will continue until the maximum count is reached. (optional)
- param: A value to pass to the DSKI to filter out unwanted data. See DSKI documentation for more details. If this parameter is not specified, the event trace comes unfiltered. (optional)
- family name: A valid family name which has been registered with DSKI. For each family, a subset of events must be specified. If all events are to be collected, the keyword all can be used instead.
- eventnames: A coma-separated list of valid event names in a family.

2. Counter

```
dstream {
        type = counter (duration = x, interval = y);
        family1 = set ( counternames ) | all;
        family2 = set ( counternames ) | all;
}
```

Parameters:

- duration: The length of experiment in seconds. (optional)
- interval: The interval between two successive readings of the counter values in seconds. (optional)
- family name: A valid family name which has been registered with DSKI. For each family, a subset of counters must be specified. If all counter values are to be collected, the keyword all can be used instead.
- counternames: A coma-separated list of valid counter names in a family.

4.2.3 Examples

The first example shows how can we use the DSKI to collect detailed event traces of the TCP/IP protocol stack. The kernel has been instrumented with several event logging points. We would like to collect event traces on the transmitting host. Specifically, we only want traces for packets going on port 42015. To do that, the following script is included in the NetSpec script.

```
dstream  machine1  {
        type = active (numevents = 500, duration = 15, params = 42015);
        ds_tcpip = set (writeSocket,  tcpOutStart,  tcpOutEnd,  ipOutStart,  ipOutEnd);
}
```

NetSpec generates the following event trace:

```
Number of events logged : 500
tcpOutStart     868657674       249080  5028            machine1
tcpOutStart     868657674       249135  5028            machine1
tcpOutStart     868657674       249167  5028            machine1
tcpOutStart     868657674       249199  5028            machine1
tcpOutStart     868657674       249234  5028            machine1
socketWrite     868657674       249929  0               machine1
tcpOutStart     868657674       250146  8100            machine1
tcpOutEnd       868657674       250188  34048002        machine1
ipOutStart      868657674       250197  34048002        machine1
ipOutEnd        868657674       250211  34048002        machine1
tcpOutEnd       868657674       250306  34049462        machine1
ipOutStart      868657674       250316  34049462        machine1
ipOutEnd        868657674       250327  34049462        machine1
tcpOutStart     868657674       250585  8100            machine1
...
```

The second experiment collects the counter values on a periodic basis. Such an experiment is useful, for example, to monitor some parameters of the connections in the experiment. For example, two counters have been defined in the ds_tcpip family to count the number of bytes sent or received. We can use the following NetSpec script to monitor those counters.

```
dstream machine1 {
        type = counter ( duration = 60, interval = 5);
        ds_tcpip = all;
}
```

NetSpec generates the following report

```
bytesIn         0          0       0
bytesOut        0          0       0
bytesIn         0          0       0
bytesOut        868657864  869704  1727908
bytesIn         0          0       0
bytesOut        868657866  877026  3274048
bytesIn         0          0       0
bytesOut        868657868  873658  4590968
bytesIn         0          0       0
bytesOut        868657870  913602  6014468
bytesIn         0          0       0
bytesOut        868657872  907498  7535788
bytesIn         0          0       0
bytesOut        868657874  911999  9051268
bytesIn         0          0       0
bytesOut        868657876  900927  10346288
```

4.3 SNMP Daemon (nssnmpd)

The function of the SNMP daemon is to act as an SNMP manager that collects a set of managed information from an SNMP agent of a network device. The SNMP daemon establishes a communication channel with a SNMP agent through which it can send and receive SNMP Protocol Data Unit(PDU). Much of the management and monitoring operations involve a periodic collection of SNMP variables. We can set up the daemon so that it periodically polls the SNMP agent of a specific network device and generates the collected information on the fly. A report collector server can then use the information to provide the latest state of that network device.

The NetSpec script defines the network device and the managed objects to monitor. The SNMP daemon then sends the SNMP PDU(s) to the SNMP agent for that particular network device and waits for the response. If a reporter channel is specified for this daemon, it directly sends the data to the reporter for further processing. Otherwise, the daemon will store the data in a local buffer until it enters the report phase in which it will send the data to the user.

The SNMP daemon uses CMU-SNMP-v1 library from Carnegie Melon University [**http://www.net.cmu.edu:80/projects/snmp/**]. The library provides basic API to create and interpret SNMP PDU as well as for creating communication channel with an external SNMP agent. The library also requires an SMI (Structure of Management Information) which includes the descriptions of the managed objects. The SMI provides the mapping between a variable name and its OID (Object IDentifier). For objects that are not included in the SMI, the full path needs to be specified in the script.

4.3.1 NetSpec Script

```
snmp machinename  {

        device =  IPADDRESS;
        mibfile = STRING;
        type = periodic ( duration = INT, period = INT);     /* If we want to monitor
                                                              *  continuously for sometime
                                                             */
                        or
        type = oneshot;
        version = v1(community = STRING);
        operation = get(var = STRING, var = STRING, …);
}
```

Table 2 Description of the parameters

PARAMETER	DESCRIPTION	VALUE SYNTAX
Device	Network device to be monitored	IPADDRESS
Mibfile	The absolute path to the SMI	STRING
Type	The type of collection	Oneshot Periodic(duration = INT, period = INT)
Version	SNMP version to use	v1(community = STRING)
Operation	SNMP operation to perform and variable names	get([var = STRING])

4.3.2 Example

If we want to poll the number of packets transmitted and received at the TCP layers every 10 seconds for 5 minutes and send the report, we can describe that experiment in the following script:

```
snmp plato  {
        device = mauchly.ittc.ukans.edu;
        mibfile = "/usr/local/lib/mib.txt";
        type = periodic(duration = 300, period = 10);
        version = v1(community = public);
        operation = get (var = "tcp.tcpInSegs", var = "tcp.tcpOutSegs");
}
```

4.4 CORBA Daemon (nscorbad)

In order to measure the performance of CORBA objects, special purpose objects called Performance Measurement Objects (PMO) [http://hegel.ittc.ukans.edu/projects/corba/pmo/index.html] were created. These objects can have a variety of behaviors, and are capable of exchanging a wide range of CORBA data with each other. The operations can be classified as full blast invocations and request response invocations wherein the sender blocks for an acknowledgement from the receiver. Measurements can be made at each PMO. Writing simple NetSpec scripts can control the behavior of these objects.

The DSKI is used to collect data, which reflects the amount of time spent in the kernel when making object request calls to and from the ORB. Using this information, a detailed time trace of activities happening inside the kernel can be obtained while the CORBA traffic is being generated. Since the DSKI already works under the control of NetSpec's integrated framework, all data (network performance and kernel performance) can be collected at one point making analysis easier.

Table 3 List of parameters in the Script and their Description

Parameter	Description
NameOfORB	ORB involved in the test
TypeOftest	Type of CORBA traffic
TestParams	Parameters needed for the test setup
Numsamples	Number of samples used to average the results
Minsize	Minimum size of data
Maxsize	Maximum size of data
Multiples	Gives the multiples in which the data size needs to be increased
Stepsize	Gives the step size in which the data needs to be increased
Predelay	Idle time before the experiment is begun
Postdelay	Idle time before the connection is torn down
Duration	Duration for which fixed size data needs to be sent
Protocol	Protocol used for Object Communication
Objname	Name of the Object
role	Role of the object in the test
Relations	Peergroup of the object
Criteria	Performance parameter that is desired
qos	QoS information
own	Network information

Table 4 List of Test Types

Type	Description
fbchar	Full blast character sequence
rrchar	Request response character sequence
fbshort	Full blast short sequence
rrshort	Request response short sequence
fblong	Full blast long sequence
rrlong	Request response long sequence
fbfloat	Full blast float sequence
fbdouble	Full blast double sequence
rrdouble	Request response double sequence
fboctet	Full blast octet sequence
rroctet	Request response octet sequence
fbstring	Full blast string sequence
rrstring	Request response string sequence
fbmstruct	Full blast struct sequence
rrmstruct	Request response struct sequence
ftp	ftp traffic type invocation using fboctets
telnet	Telnet traffic type invocation using rroctets

4.4.1 Example

The script essentially consists of three parts – the first part is pertinent to the receiver object, the second to the router and the third part to the sender object. The sender sends packets to the receiver through the router and the performance of the ORB in terms of throughput is measured.

```
cluster {

  corba marcus {
        type = proxy  (orb=omniORB2 ,
                       numsamples = 10 , minsize = 512 ,
                       maxsize = 1048576  ,  multiples = 2 ,
                       predelay = 2 , postdelay = 0 ,
                       duration = 30 );
        protocol = iiop;
        objname = sri3;
        role = receiver;
        PeerGroup = xyz;
        criteria = throughput;
        qos      = normal;
        own      = marcus  (interface = eth , port = 33151);
```

```
 }

 corba marcus {
        type = proxy (orb=omniORB2 ,
                      numsamples = 10 , minsize = 512 ,
                      maxsize = 1048576  ,  multiples = 2 ,
                       predelay = 6 , postdelay = 3 ,
                      duration = 30 );
        protocol = iiop;
        objname = abcd;
        role = router ;
        PeerGroup = xyz;
        criteria = throughput;
        qos      = normal;
        own      = marcus (interface = eth );
 }

 corba marcus {
        type = proxy (orb=omniORB2 ,
                      numsamples = 10 , minsize = 512 ,
                      maxsize = 1048576  ,  multiples = 2 ,
                      predelay = 10 , postdelay = 2 ,
                      duration = 30);
        protocol = iiop;
        objname = sri3;
        role = sender;
        PeerGroup = xyz;
        criteria = throughput;
        qos      = normal;
        own      = marcus (interface = eth );
 }
}
```

Report

The following report was generated when the given script was executed.

```
NetSpec Version 3.0 alpha 1 Control Daemon[19303] ready
NetSpec corba Daemon marcus.ittc.ukans.edu[19304] ready
NetSpec corba Daemon marcus.ittc.ukans.edu[19305] ready
NetSpec corba Daemon marcus.ittc.ukans.edu[19306] ready

NetSpec Control daemon, NetSpec Version 3.0 alpha 1

Report from marcus.ittc.ukans.edu[19303] generated @ 03/04/99 17:39:50
CST



 omniORB2 proxy (receiver side) throughput performance results
```

```
 omniORB2 proxy (router side) throughput performance results


 omniORB2 proxy (sender side) throughput performance results
Sending Longs ...

real_time    = 476.516ms
user_time    = 80ms
system_time  = 20ms
524.641 calls/second
CPU time     = 0.1micro secs
Total Cumulative Time = 476.516ms

Sending Shorts ...

real_time    = 308.746ms
user_time    = 40ms
system_time  = 10ms
809.727 calls/second
CPU time     = 0.05micro secs
Total Cumulative Time = 785.262ms

Sending Octets ...

real_time    = 340.35ms
user_time    = 100ms
system_time  = 0ms
734.538 calls/second
CPU time     = 0.1micro secs
Total Cumulative Time = 1125.61ms

Sending Mixed(Long/Short/Octet) ....

real_time    = 956.131ms
user_time    = 120ms
system_time  = 70ms
261.47 calls/second
CPU time     = 0.19micro secs
Total Cumulative Time = 2081.74ms
```

Sending Structs ...

…

4.5 Private Network – Network Interface Daemon (pnnid)

PNNI is a topology state routing protocol in which nodes flood Quality of Service and reachability information to all other nodes so that each node obtains complete information about the topology of the network and its current state. In an effort to reduce the complexity of such a scheme, PNNI uses a hierarchical model to divide nodes into peer groups.

4.5.1 Features of PNNI

- PNNI has the capability to automatically configure itself when the address structure reflects the topology of the network
- The hierarchical model allows it to scale well for large networks.
- Crankback feature allows it to reroute around failed components during call setup
- Connection follows the same path as the setup message used.
- Uses many routing metrics including cell transfer delay, delay variation, current and average peak load.

The PNNI Simulation and Emulation tools has a common user interface designed to facilitate the user in specifying the minimal required parameters to setup a network and run experiments of varying complexity and magnitude. NetSpec is used to conduct the experiments of the PNNI Simulator/Emulator. However, in an effort to reduce the complexity of the scripts, this tool provides a wrapper that takes a simple script and creates a NetSpec script containing all the additional required information.

To perform emulation, NetSpec is used to distribute the experiment to a specified pool of workstations. NetSpec use a control daemon to communicate with PNNI slave daemon processes on target workstations, providing each of them with an individual script containing information about the network to be setup. Upon completion, all the PNNI daemons report the experimentation results to the control daemon at the main workstation, which are printed on the console.

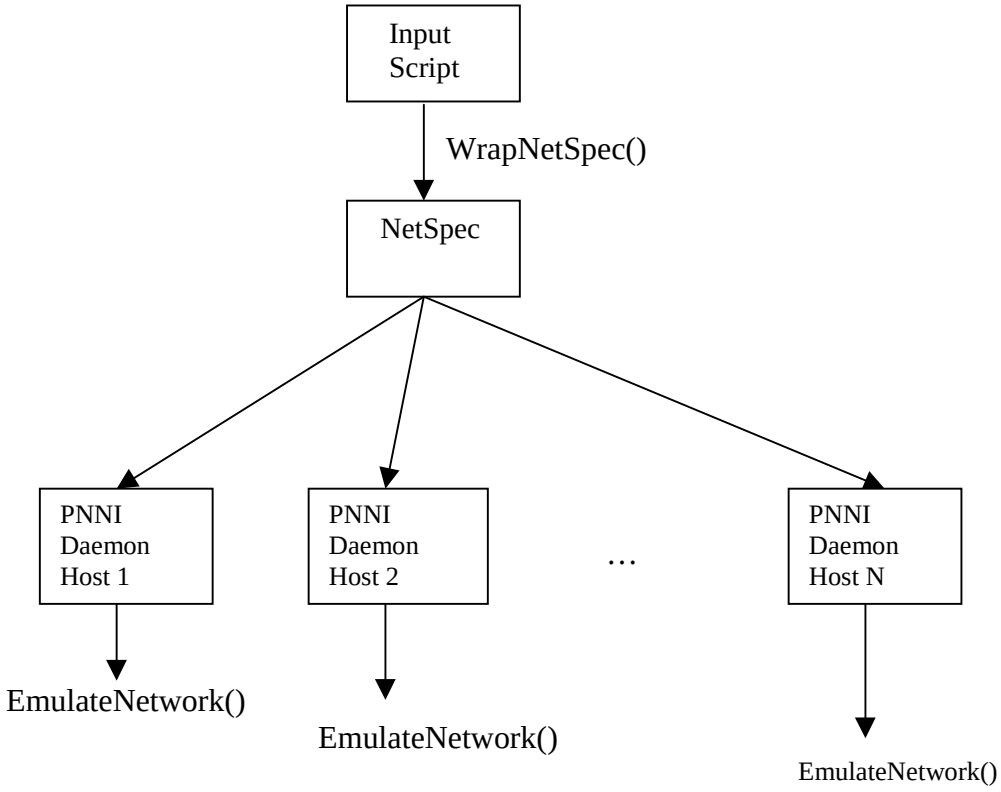

Figure 5 Architecture of a PNNI Emulation

4.5.2 NetSpec Script

The cluster keyword tells NetSpec that the end points enclosed should execute in parallel, but are also somehow related to each other. The pnnid keyword tells NetSpec which daemon to contact. Following pnni is the workstation on which the daemon is to be contacted and after the colon is the TCP port on which to contact it. The information between the braces following the port number is the information that NetSpec will parse for conformity with its requirements and pass along to the daemon.

NetSpec is thus invoked and furnished the script produced by the wrapper. NetSpec contacts the PNNI daemons on the specified hosts and passes them the relevant portion of the script. A PNNI daemon, like any other NetSpec daemon is responsible for parsing the information that the control daemon gives it.

Each pnni daemon then begins to run its part of the experiment. It first forks off a child process for each virtual node and then for each virtual host on its machine. The parent then waits for its children to finish. Each child then goes on to link itself with the nodes or hosts with which it is supposed to connect and the experiment proceeds over TCP sockets and in real time.

```
1 node = genericnode (duration=150, core_nodes=2, edge_nodes=2,
2               prop_constant=25, flooding_threshold=5,
3               numports=4, core_process_time=3.0,
4               utiln_log_period=20, queuesize=1000);

5 host = generichost (duration=150, calls=15, arrival_period=5,
6           bw=1000, duration_period=45, queuesize=1000,
7           host_process_time=3.0);
8 host = H3 (dest_fixed=0);
9 host = H0 (dest_fixed=3);

10 port = genericport (bw=OC12, delay=10);

11 connection = E0->C1;
12 connection = C1->C2;
14  connection = C2->E3;
```

A Sample of the Result Produced

```
SimKernel: Simulation ended in sim time 000150.000000
Duration of simulation is 2 seconds
Type CTRL-C to end the process. Thanks.
Avg  PROCESSING TIME 000000.000566
Over Total events 4435

***** CALL SETUP LOGS START *****

-- H0 host record begins -----------------------------

calltype  bw(kbps)    starttime         stoptime       setuptime
result   cause
cbr         999.792 00:00:15.000  00:00:15.197  000.197000  setup
cbr         999.792 00:00:20.000  00:00:20.126  000.126000  setup
cbr         999.792 00:00:25.000  00:00:25.139  000.139000  setup
cbr         999.792 00:00:30.000  00:00:30.132  000.132000  setup
cbr         999.792 00:00:35.000  00:00:35.139  000.139000  setup
cbr         999.792 00:00:40.000  00:00:40.126  000.126000  setup
cbr         999.792 00:00:45.000  00:00:45.138  000.138000  setup
cbr         999.792 00:00:50.000  00:00:50.126  000.126000  setup
cbr         999.792 00:00:55.000  00:00:55.139  000.139000  setup
cbr         999.792 00:01:00.000  00:01:00.132  000.132000  setup
cbr         999.792 00:01:05.000  00:01:05.139  000.139000  setup
cbr         999.792 00:01:10.000  00:01:10.126  000.126000  setup
cbr         999.792 00:01:15.000  00:01:15.138  000.138000  setup
cbr         999.792 00:01:20.000  00:01:20.126  000.126000  setup
cbr         999.792 00:01:25.000  00:01:25.139  000.139000  setup

total cbr calls          : 15
% successfull cbr calls  : 100
total cbrbw request(mb)  : 14.9969
```

```
cbrbwrej(mb)             : 0
mean callsetup time      : 000.137466

-- H0 host record ends --------------------------------

-- H3 host record begins ------------------------------

calltype  bw(kbps)    starttime         stoptime       setuptime
result   cause
cbr         999.792 00:00:15.000  00:00:15.198  000.198000  setup
cbr         999.792 00:00:20.000  00:00:20.126  000.126000  setup
cbr         999.792 00:00:25.000  00:00:25.139  000.139000  setup
cbr         999.792 00:00:30.000  00:00:30.132  000.132000  setup
cbr         999.792 00:00:35.000  00:00:35.139  000.139000  setup
cbr         999.792 00:00:40.000  00:00:40.126  000.126000  setup
cbr         999.792 00:00:45.000  00:00:45.138  000.138000  setup
cbr         999.792 00:00:50.000  00:00:50.126  000.126000  setup
cbr         999.792 00:00:55.000  00:00:55.139  000.139000  setup
cbr         999.792 00:01:00.000  00:01:00.132  000.132000  setup

total cbr calls          : 15
% successfull cbr calls  : 100
total cbrbw request(mb)  : 14.9969
cbrbwrej(mb)             : 0
mean callsetup time      : 000.137533

-- H3 host record ends --------------------------------

AVG RESULTS OF ALL CALLS

total cbr calls             : 30
% successfull cbr calls     : 100
total cbrbw request(mb)     : 29.9938
cbrbwrej(mb)                : 0
mean callsetup time         : 000.137500

***** CALL SETUP LOGS END *****

##### NODE INSTRUMENTATION LOGS START #####

-- C1 node record begins ------------------------------

convergence time              : 000000.140000

...
```

4.6 Switch Daemon (nsswitchd)

This NetSpec daemon was designed to configure the switch. In the run phase , it opens the software switch device driver, /dev/atmsw, and executes ioctl's to setup the connections and set the switch parameters as specified in the script. It does NOT generate a report.

4.6.1 NetSpec Script

```
cluster {
        swswitch testbed4:54321 {
          type = switch (MaxQueueLength=1000,
                         LinkCapacity=6500000,
                         ABRAlgorithm=EPRCA,
                         port0_rtt=10000);
          type = connection ( VCtype=PVC,
                              inport=1, inVPI=0, inVCI=40,
                             outport=2, outVPI=0, outVCI=40);
        }
}
```

The two allowable types are "switch" and "connection". Below are the valid parameters for each.

4.6.2 Description of Parameters

Table 5 Allowable Parameters of parameter type " switch "

Parameter	Description
MaxQueueLength = <INT>	Specifies the length of the queues on the switch. This version of the software switch does per-class queuing and two classes are used (VBR and ABR).
SwitchPeriod = <INT>	The software switching function is a KURT (KU Real-Time) module that is invoked on a periodic schedule. This specifies the period in microseconds.
LinkCapacity = <INT>	This specifies the capacity of the links on the switch in Mbps.
CellsPerPacket = <INT>	To increase the throughput of the switch, it was modified to switch AAL5 packets into which we can stuff several AAL0 cells. This gave the illusion of higher cell-level throughput. This parameter specifies the number of AAL0 cells to pack into AAL5 packets.

ABRAlgorithm = ERICA	This parameter has only one allowable value as of now - the literal "ERICA". It is an algorithm to calculate available bandwidth on the switch for ABR. There are other such algorithms, such as EPRCA, but they have not been implemented as of yet.
PORT<INT>_RTT = <INT>	This parameter allows the user to set the simulated round-trip delay for a cell or packet on a particular port. Here, only 3 ports are allowed per switch (0, 1, and 2). The values are in milliseconds.

Table 6 Allowable Parameters of parameter type " connection "

Parameter	Description
Vctype = PVC || SVC	This specifies whether the connection is a switched virtual circuit or a permanent virtual circuit.
inport = 0 || 1 || 2	This is the ENI interface number (port) for the "in" side of the connection. It is limited to 0, 1, or 2 due to the limitation that placed on the software as described above.
inVPI = <INT>	This is the virtual path identifier for the "in" side of the connection. Although any integer is accepted as valid input, Linux-ATM will only accept a VPI of zero at present.
inVCI = <INT>	This is the virtual circuit identifier for the "in" side of the connection. Although any integer is accepted as valid input, VCIs 1 to 32 are reserved and therefore, not valid.
outport = <INT>	This is the ENI interface number (port) for the "out" side of the connection. It is limited to 0, 1, or 2 due to the limitation that placed on the software as described above.
outVPI = <INT>	This is the virtual path identifier for the "out" side of the connection. Although any integer is accepted as valid input, Linux-ATM will only accept a VPI of zero at present
outVCI = <INT>	This is the virtual circuit identifier for the "out" side of the connection. Although any integer is accepted as valid input, VCIs 1 to 32 are reserved and therefore, not valid

ALL of these parameters are REQUIRED to setup a connection and they are expected in the order presented, separated by commas as in the example script.

4.7 Host Daemon (nsabrd)

This daemon is merely an extension to the Test daemon. The Test daemon has been modified to send ABR traffic over ATM. The Test daemon has essentially been modified to work with ATM at the link level by establishing ATM SVCs or PVCs as specified by the user

4.7.1 NetSpec Script

For PVCs:

```
cluster {
          atmtest testbed5:54321 {
            type = atmFull (cellsPerPdu=1,totalblocks=8000, blocksize=1000,
                            traffictype=ABR, PCR=8000, ICR=1000, MCR=0);
           protocol = atm (PVC, intf=0, vpi=0, vci=80);
           own = testbed5;
           peer = testbed6;
          }
        atmtest testbed6:54321 {
          type = atmSink (totalblocks=8000, blocksize=1000,
                          traffictype=ABR, PCR=8000, ICR=1000, MCR=0);
         protocol = atm (PVC, intf=0, vpi=0, vci=60);
        own = testbed6;
        peer = testbed5;
      }
}
```

For SVCs:

```
cluster {
          atmtest testbed5:54321 {
            type = atmFull (cellsPerPdu=1,totalblocks=10000, blocksize=1000,
                            traffictype=ABR, PCR=8000, ICR=4000, MCR=2000);
           protocol = atm (SVC, local_switch=micro_switch,
                           remote_switch=micro_switch, no_of_connections=1);
           own = testbed5;
           peer = testbed6;
          }
        atmtest testbed6:54321 {
          type = atmSink (totalblocks=10000, blocksize=1000,
                          traffictype=ABR, PCR=8000, ICR=4000, MCR=2000);
          protocol = atm (SVC, local_switch=micro_switch,
                          remote_switch=micro_switch, no_of_connections=1);
          own = testbed6;
          peer = testbed5;
```

```
    }
}
```

Since this daemon is similar to the test daemon it contains the full parameter set of the original nstestd plus the additions below.

The two allowable types for ATM are "atmFull" and "atmSink". They have identical parameter sets. Below are the valid parameters for both.

Table 7 Allowable Parameters for "atmFull" and "atmSink"

Parameters	Description
cellsPerPdu = <INT> totalblocks = <INT> blocksize = <INT>	These options are common to those that NetSpec already uses for traffic tests.
traffictype = ABR | CBR	This specifies the type of traffic to generate. CBR is constant bit rate, ABR is available bit rate.
PCR = <INT>	This is the peak cell rate for the connection in cells per second.
ICR = <INT>	This is the initial cell rate for the connection in cells per second. The connection will start at this rate.
MCR = <INT>	This is the minimum cell rate for the connection in cells per second.

This daemon adds the ATM protocol type, which has two valid parameter sets - one for PVCs and one for SVCs. The two are shown below.

Table 8 Parameter Set for PVC

Parameters	Description
PVC	This specifies that it is indeed a PVC connection.
intf = 0 | 1 | 2	This is the ENI interface number (port) for the "in" side of the connection. It is limited to 0, 1, or 2 due to the limitation placed on the number of NICs as described above
vpi = <INT>	This is the virtual path identifier for connection. Although any integer is accepted as valid input, Linux-ATM will only accept a VPI of zero at present.
vci = <INT>	This is the virtual circuit identifier for the connection. Although any integer is accepted as valid input, VCIs 1-32 are reserved and therefore, not valid.

ALL of these parameters are REQUIRED to setup a connection and they are expected in the order presented, separated by commas as in the example script.

Table 9 Parameter Set for SVC

Parameters	Description
SVC	This specifies that it is indeed a SVC connection.
local_switch = <IDENTIFIER>	This is the name of the local switch. If the local switch is a software switch, then "micro_switch" is specified.
remote_switch = <IDENTIFIER>	This is the name of the remote switch. If the remote switch is a software switch, then "micro_switch" is specified.
no_of_connections = <INT>	This specifies the number of SVCs to create.

ALL of these parameters are REQUIRED to setup a connection and they are expected in the order presented, separated by commas as in the example script.

It generates a report almost identical (maybe identical) to the nstestd's report.

5.0 Appendix

5.1 Installation of NetSpec

Copy the files from the tar supplied netspec/bin directory into /usr/local/bin. (You may change this location if you prefer. If you change locations, then you will need to edit the supplied netspec.conf file and modify the inetd.conf and services line additions as necessary).

Copy netspec.conf into /etc.

NetSpec can be installed in either of the two ways described below :

5.1.1 inetd Installation

Edit /etc/services and add the following line:

```
netspec 42003/tcp # NetSpec Daemon
```

and then add the following line to /etc/inetd.conf:

```
netspec stream tcp nowait root /usr/local/bin/netspecd netspecd
```

Finally, you have to reinitialize inetd, so determine the process id of inetd ("ps axuw | grep inetd" on BSD-ish stuff, "ps -elf | grep inetd" on systemV-ish) and perform a "kill -1" on this process. You might choose to install the daemons owned by nobody, but this restricts certain options such as changing MTU sizes.

5.1.2 Standalone Installation (i.e. you don't have root access)

Place the executables at the destination of your choice, add them to your path environment variable for ease of use, or cd to the dir and execute netspecd with the "-s" option on all the machines that are being used for the test. This will start the daemon in standalone mode. Every time the systems reboot you will need to restart them. In case you don't have access to /etc, you can alternatively specify the -c option to netspecd. All daemons have a help function, namely -h.

5.2 Working of NetSpec

On hosts on which NetSpec has been installed we find the controller (nscntld), the netspecd, a test/measurement daemon (for e.g., nstestd) and the executable (netspec) being necessary to run performance measurement tests. The netspecd is always running at 42003. To analyze how the daemons operate consider the following script:

```
cluster  {

        test lovelace : 42030 {
                type = full (blocksize=32768, duration=30);
                protocol = tcp (window=32768);
                own = lovelace.atm:42020;
                peer =marr.atm:42020;
 }

        test marr :42040 {

           type = sink (blocksize=32768, duration=30);
           protocol = tcp (window=32768);
           own = marr.atm:42020;
           peer = lovelace.atm:42020;
 }
 }
```

This script is run as follows:

galaga % netspec tcp.single

This command contacts the netspecd, which is running at 42003, and it gives it the keyword *cluster*. The netspecd looks into its netspec.conf and finds that it has to start up the controller (nscntld). This in turn parses the script and finds that it has to invoke the test daemons (nstestd) corresponding to the keyword *test*. The controller contacts the netspecd running in hosts marr and lovelace and the netspecd in turn starts up the nstestd in both the hosts and the controller shelves off the appropriate parts of the script to the test daemons. From this point the test daemons parse the script and based on the commands from the controller i.e., setupCommand, openCommand, runCommand, etc they run the tests and generate the report back to the controller.

If the script is modified as given in italics i.e., it is modified to make the daemons run in standalone mode. Here the nstestd is run in standalone mode at port 42030 in host lovelace and similarly at port 42040 in host marr. In this case the controller contacts the nstestd running at the specified ports instead of going through the netspecd. Similarly the nscntld can be run in standalone mode by specifying the machine:port after the keyword cluster.

5.3 Tables

5.3.1 Table 10 Explanation of NetSpec Script keywords

KEYWORDS	DESCRIPTION	ALTERNATIVES
Parallel	Indicates parallel multiple traffic connections. It can be point-to-multipoint or multipoint-to-point connections. Data flows between the source and receiver of each block simultaneously	Serial or nothing to indicate single connection
serial	Indicates serial multiple traffic connections. Data flows between the source and receiver of each block after the previous data flow is completed. (one at a time)	Parallel or nothing to indicate single connection
cluster	Indicates the beginning of a structured block, where the source and destination end systems will be defined, as well as the various network parameters. If a machine domain name or its IP follows (after) then the control daemon (nscntrld) will run on that machine. If nothing follows, then the control daemon will run in the machine you are running the NetSpec script.	NONE
test	Specifies where the test daemon (nstestd) will run. Apparently it specifies the end systems under test. The end system domain name or its IP address can be passed as parameters	NONE
type =	Indicates in what mode (type) the experiment will run. Fullstream, burst, or queued burst types can passed as parameters.	NONE
full	The test will run on the full stream mode. That means that the end systems will transmit data as fast as possible	burst, burstq
Blocksize =	Indicates the burst size in blocks(bytes).	NONE
duration =	Indicates the duration of your experiment in seconds. Tests in the University of Kansas showed that NetSpec can run successfully for hours.	NONE
period =	Indicates the burst period in microseconds. This is only used in the	In the full stream mode, this parameter is

	burst and queued burst modes	omitted
protocol =	Indicates the transport protocol to be used in the experiment	NONE
tcp	The TCP protocol is used throughout the experiment	udp
udp	The UDP protocol is used throughout the experiment.	tcp
window=	The TCP window size in bytes to be used in the experiment	When UDP is used, this parameter is omitted
own=	Indicates the local machine (by domain name or IP address) where the test daemon is running within the block. A machine port can be identified as well if desired by the user. Therefore, in the script source block (the block where after "type=", "sink" is not following) this keyword is followed by the transmitter, whereas in the script receiving block (the block where after "type=", "sink" follows) this keyword is followed by the receiver.	NONE
peer=	Indicates the local machine (by domain name or IP address) where the test daemon is running within the block. A machine port can be identified as well if desired by the user. Therefore, in the script source block (the block where after "type=", "sink" is not following) this keyword is followed by the receiver, whereas in the script receiving block (the block where after "type=", "sink" follows) this keyword is followed by the transmitter	NONE

5.3.2 Table 11 Keywords used in Emulated Traffic Scripts

Keywords	Description	Required Parameters
TelnetSessionInterarrival	Interarrival Time for TELNET sessions	lambda
TelnetSessionDuration	Duration of a TELNET session	mean, stdDeviation
TelnetPacketInterarrival	Interarrival of TELNET packets in a TELNET session	Fixed Model
TelnetPacketSize	TELNET Packet size	Fixed Model
FtpSessionInterarrival	Interarrival Time for FTP sessions	lambda
FtpNOfItems	Number of Items transfered in an FTP session	Fixed Model
FtpItemSize	Size of FTP items in an FTP session	Fixed Model
VoiceSessionInterarrival	Interarrival Time for Voice sessions	lambda
VoiceSessionDuration	Duration of Voice sessions	lambda
VideoTeleConferenceFrame Size	Frame Size of a Video Teleconference Session	scale, shape
VideoMPEGFrameSize	Frame Size of a Video MPEG Session	sceneLengthMean, Imean,IstdDeviation, Pmean, PstdDeviation, Bmean,BstdDeviation
WWWRequestInterarrival	Interarrival Times for WWW Requests	lambda
WWWItemSize	Size of Items transferred in WWW Request	Fixed Model

6.0 Bibliography

[1] Roelof J.T. Jonkman. NetSpec: Philosophy, Design and Implementation. Master's Thesis, University of Kansas, Informations and Telecommunications Sciences Laboratory.

[2]Beng-Ong Lee. Wide Area ATM network experiments using Emulated Traffic sources. Master's Thesis, University of Kansas, January 1998

[3]Anil Gopinath. Performance Measurement and Analysis of Real-Time CORBA Endsystems, Master's Thesis, University of Kansas.